Communicating Process Architectures 2005
Jan Broenink, Herman Roebbers, Johan Sunter, Peter Welch, and David Wood (Eds.)
IOS Press, 2005

`jcsp.mobile`: A Package Enabling Mobile Processes and Channels

Kevin CHALMERS and Jon KERRIDGE
School of Computing, Napier University, Edinburgh EH10 5DT
{k.chalmers, j.kerridge}@napier.ac.uk

Abstract. The JCSPNet package from Quickstone provides the capability of transparently creating a network of processes that run across a TCP/IP network. The package also contains mechanisms for creating mobile processes and channels through the use of filters and the class Dynamic Class Loader, though their precise use is not well documented. The package `jcsp.mobile` rectifies this position and provides a set of classes and interfaces that facilitates the implementation of systems that use mobile processes and channels. In addition, the ability to migrate processes and channels from one processor to another is also implemented. The capability is demonstrated using a multi-user game running on an ad-hoc wireless network using a workstation and four PDAs.

Keywords. JCSP, JCSPNET, Mobile Processes and Channels, Dynamic Class Loading, Migratable Processes and Channels, Location Aware Computing

Introduction

The motivation for the work reported in this paper came from the desire to modify an existing multi-player maze game that ran on a network of PC workstations to a version that used PDAs for the player processes and a wireless enabled laptop for the display process. The initial reason for building the game was to demonstrate the use of JCSPNet [1] in a manner that would motivate students to better understand the concepts of concurrency, parallelism and the benefits of the JCSP approach. In the event, the project went far beyond this initial goal and resulted in the creation of a package the enables the design and implementation of systems that use mobile processes and channels. The JCSPNet package already contained the basic mechanisms for achieving this but was somewhat tortuous in the way that such processes were created and incorporated into a complete system. The `jcsp.mobile` package brings together a number of the existing JCSPNet features and capabilities into a framework that makes such systems easier to design, implement and use.

In the next section we shall present the justification for the design of `jcsp.mobile` and describe the main components from which it is constructed. Section 2 will describe the relevant mobile parts of the maze game and it will also show how the concepts can be utilized in the implementation of a home consumer electronics remote control that requires just one controller for all such devices. In the next section we shall show how *migratable* processes and channels can be implemented using a simple producer-consumer based example. Finally, the package is evaluated, some conclusions are drawn and areas for future work identified.

1. Overview of `jcsp.mobile`

JCSPNet does include classes and interfaces associated with *Dynamic Class Loading*. However the API documentation provided does not give any indication of how these classes and interfaces should be used. Initial attempts to understand the problem involved sending objects that implemented the CSProcess interface across channels and though initially this gave the impression of working it soon became apparent that the problem was far more complicated than such a simple approach permits. It is in fact necessary to understand how Java loads classes and any other classes contained within a class that is executed on a processor and furthermore to understand the problems that can occur if the designer does not fully appreciate the intricacies of the mechanism. Given that the ability to transmit Applets around the web was one of the design principles of Java it is somewhat surprising that this capability is so opaque.

The `jcsp.mobile` package creates a new concept called a Process Channel which hides both the class `DynamicClassLoader` and the class `Filter` [1] that has to be added to a channel so that it can communicate class objects. The various types of Process Channel are defined as interfaces within Mobile and instances of these Process Channels are created in a manner analogous to ordinary channel creation in a *Channel Name Server* (CNS) [1]. An abstract class `MobileProcess` is provided which the user simply extends to create their own mobile process. A mobile device has to run a process called `MobileProcessClient` that initializes itself with a call to `Mobile.init()` as a node in a mobile network, which then gives it access to all the facilities of the `jcsp.mobile` package. The initialization mechanism mimics that used to initialize a node of an ordinary CNS style network.

Mobile processes within CSP are first described within the occam language by Barnes as a proposal [2], after developing some of the necessary functionality required for the language. This proposal, although more applicable for a different platform, can be tailored for use within JCSP, particularly the concept of one process (a server) transferring another process (the mobile process) to another process (the client). Within the Java language, the functionality that required development for the occam language is not necessary, because Java uses references to objects (if an object is said to equal another, only one such object exists) as opposed to occam's use of copy semantics (when an object is said to equal another, two unique objects are in existence) [3] as well as the Java language's interpreted nature

1.1 Dynamic Class Loading

Due to the interpreted nature of the Java language, it is possible to load new classes into the runtime environment as required by the system. Basically, this means that it is possible to send a class description to an executing process (e.g. read from a file, sent over a network) and for that process to create new instances of the class as required. How this works is not clearly described, although there are some attempts [4] that claim a clear definition , mentioning the already poor documentation issues. However, these do not accurately give a description either.

Figure 1 illustrates how class loading operates within Java. In this system, three class loaders are defined, the System Class Loader (started when the JRE starts) and two developed class loaders, Class Loader 1 and Class Loader 2. These both have the System Class Loader as a parent. These developed class loaders have information relating to loading specific classes, and if they cannot load a class, they call the System Class Loader to load it for them. The System Class Loader has no knowledge of its child class loaders.

Looking at each of the class loaders in turn, the System Class Loader knows about Class A and Class B and can only use these classes within its context. For example, if Class A were to try and create an instance of Class C or Class D, an exception would be raised relating to the class definition not being found.

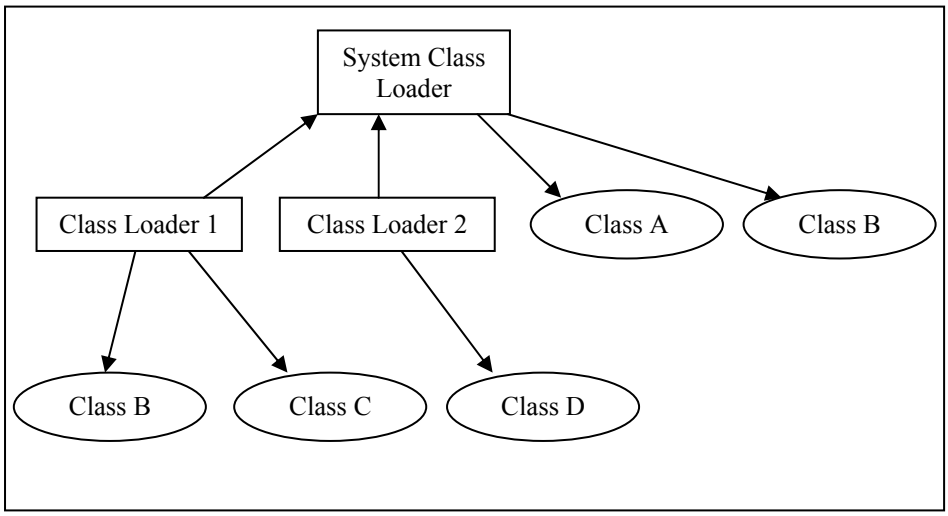

Figure 1 Class Loading in Java

For Class Loader 1, as well as the classes available from the System Class Loader, and Class C, a further version of Class B is available. This can lead to versioning problems between classes, i.e. Class C may expect the system defined version of Class B yet receive the one defined by Class Loader 1. This may also occur with Class A as well. Class A will use the System Class Loader to load itself and any classes within itself, and the problem of a Class Loader 1 version of Class B being passed into Class A and causing the versioning problem can arise.

Class Loader 2 only knows how to define Class D and the system classes, and therefore any attempt to create an instance of Class C will fail with the class not defined exception. If Class D should need instances of Class A or Class B, it would call upon the System Class Loader to load them.

Classes are generally defined using an array of bytes equivalent to the data normally stored within the class file. These bytes can obviously be sent across a communications medium such as the channels being used within JCSP, and initially this was the route taken when examining sending class data with an object. It is hinted at, however, that it is possible to set channels to send this data automatically [5], although how is not well documented. This led to an investigation as to how to achieve class loading over channels.

The first point of investigation was the documentation available for the package. The two overview documents provide little in depth discussion of how to use anything but the most basic features, and suggests the API specification as a point of reference for more advanced features. The problem here is that API specifications provide descriptions of all the classes available within the package, and have one starting access point available. It is a matter of guess work where to start the search from here. Examination of the factory methods associated with channel creation and the specialization required to create these class-loading channels was fruitless. In the end a text search for “class loading” in the documentation located a starting point.

This, however, did not clear up how to implement class loading easily, and still expected some further knowledge before it could be used. What the documentation describes is the creation of service - an instance of `DynamicClassLoader` - that needs to be

installed onto a Node using its Service Manager. This simply runs services available on the Node and is of no concern here. When the class loading service is started, it creates two new processes a JFTP Server (there is no mention what JFTP stands for) and a Class Manager. Exactly how these operate is not described. It was deduced that when an object is sent across the channel, the Class Manager attempts to load the class data from its available class definitions. If it should fail it will request the class definition from the sending process's JFTP, and when received, uses this to load the class data for the object it has just received.

Although the creation and running of the `DynamicClassLoader` service may appear to solve the problem, the little documentation available does mention attaching filters to the channel ends to allow the sending and receiving of class data down the declared channels. This led an investigation of exactly how filtered channels operate, and fortunately the documentation for this feature is more concise.

JCSP allows the creation of a `FilteredChannel` object from other channel ends (`FilteredChannelInput` and `FilteredChannelOutput` respectively) and then the addition of filters to these `FilteredChannel` objects as required. This channel type has the same functionality externally as the channel used to create it. Internally, when the relevant operation (read or write) is called, it uses the filters added to it to modify the message in some way. What happens with the Dynamic Class Loading filter is that the location of the sending Node's JFTP is attached. When the receiving Node reads the object from the channel, the Class Manager is passed this location. Then, if class definitions are required for the object received, the Class Manager can request it from the sending location's JFTP. This leads to the breakdown of Dynamic Class Loading given in Figure 2.

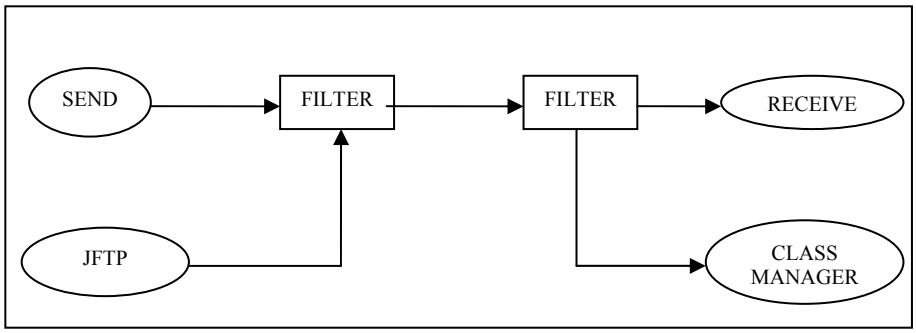

Figure 2 Dynamic Class Loading Using Filters

This model demonstrates how the Filter adds the JFTP location to the object being sent by the Send process, and then sending it down the channel. At the receiving end, the Filter takes the message, extracts the JFTP location and sends it on to the Class Manager. The original object is passed to the Receive process, which can then use the object, even if it had no prior knowledge as to what the object was.

Although this may seem quite a trivial concept when described here (start a service, add filter to the channel then read from channel as normal), the lack of good documentation created a situation where much searching and experimentation was involved. This could have been avoided. However, the simplicity of the approach does mean it can be started and used easily, with little modification to an existing system. One of the primary reasons for this complexity is to deal with the case where a process uses other classes within its definition and it would not be immediately obvious that these other classes would be required. The mechanism implemented ensures that such additional class definitions are obtained automatically.

The result of this experimentation is a package, `jcsp.mobile`, the components of which are now described.

1.2 Mobile

The class `Mobile` captures the basic functionality of the mobile capability. It defines a `DynamicClassLoader` {2}[1] and it is presumed that only one such instance per node of Mobile is initialized, in a similar manner to the creation of CNS in JCSPNet. A `ServiceManager` {8, 9} is created into which `theClassLoader` is installed {10}, named and then started in the method `init` {5-17}. The `NodeKey` for the `init` method is generated by a call to the `TCPIPNodeFactory` in a similar manner to that used by CNS {8}.

```
01  public class Mobile  {
02    protected static DynamicClassLoader theClassLoader;
03    private static boolean isInitialised = false;
04
05    public static void init(NodeKey thisNode) {
06      if (!isInitialised) {
07        theClassLoader = new DynamicClassLoader();
08        ServiceManager theServiceManager = Node.getInstance().
09                                          getServiceManager(thisNode);
10        theServiceManager.installService
11                                  (theClassLoader, "Class Loader");
12        theServiceManager.startService("Class Loader");
13        isInitialised = true;
14      }
15      else
16        System.out.println("Mobile already initialised");
17    }
18
19    public static ProcessChannelOutput createOne2Net (
20                          NetChannelLocation channelLocation)
21    {
22      return ProcessChannelEndFactory.
23                               createOne2Net(channelLocation);
24    }
25    public static ProcessChannelOutput createOne2Net
26                                        (String channelName)
27    {
28      return ProcessChannelEndFactory.createOne2Net(channelName);
29    }
30    public static ProcessChannelOutput createOne2Net
31                                        (NetChannelOutput netOut)
32    {
33      return ProcessChannelEndFactory.createOne2Net(out);
34    }
35    ...  plus other shared output and input variants
36    public static AltingProcessChannelInput createNet2One()
37    {
38      return ProcessChannelEndFactory.createNet2One();
39    }
40    public static AltingProcessChannelInput createNet2One
41                                              (String channelName)
42    {
43      return ProcessChannelEndFactory.createNet2One(channelName);
44    }
45    public static AltingProcessChannelInput createNet2One
46                                        (NetAltingChannelInput netIn)
47    {
48      return ProcessChannelEndFactory.createNet2One(in);
49    }
50  }
```

[1] The notation {n} in the text refers to a line number in a program listing

The static methods of `Mobile` can then be called in manner analogous to CNS to create various types of Process Channels. Each of these static calls results in the calling of a similar method in a `ProcessChannelEndFactory`. Some of these static methods are shown {19-49}. Those omitted simply create the shared versions of the channels. Three versions of the channel creation method are provided, the first takes a `NetChannelLocation` parameter, used in the creation of anonymous channels or when a *channel end* is communicated over a channel {22}. The second takes a `String` parameter that is used in the same manner as creating a named network channel using the CNS {28}. The third creates a Process Channel from an existing Net Channel {33}, which is necessary to allow simple creation of Migratable Channels with class loading capabilities (see section 3).

1.3 Process Channels

The Process Channels are simply an extension of the existing `NetChannel` interfaces provided within JCSPNet. In order that we can use `ProcessChannelInputs` in an alternative, an abstract class `AltingProcessChannelInput` is created {58-69} that provides the required functionality.

```
51  public interface ProcessChannelInput extends NetChannelInput
52  {
53  }
54  public interface ProcessChannelOutput
55                                      extends NetChannelOutput
56  {
57  }
58  public abstract class AltingProcessChannelInput
59                                extends NetAltingChannelInput
60                               implements ProcessChannelInput
61  {
62    public AltingProcessChannelInput()
63    {
64    }
65    public AltingProcessChannelInput(AltingChannelInput alt)
66    {
67      super(alt);
68    }
69  }
```

The `ProcessChannelEndFactory` provides factory methods for creating instances of the Process Channels using the same method names as those used in the CNS. These are the methods called from the singleton class `Mobile` described previously. Variants are provided that manipulate a `NetChannelLocation` {72}, or will invoke a call to the corresponding CNS method using a character string identifier for the channel being created {76}, or create a Process Channel from an existing Net Channel {79}.

```
70  class ProcessChannelEndFactory {
71    protected static One2Process createOne2Net(
72                                NetChannelLocation channelLocation){
73      return new One2Process(channelLocation);
74    }
75    protected static One2Process createOne2Net(String channelName){
76      return new One2Process(channelName);
77    }
78    protected static One2Process createOne2Net
79                                          (NetChannelOutput netOut){
80      return new One2Process(netOut);
81    }
```

```
82
83    protected static Process2One createNet2One(){
84      NetAltingChannelInput netIn = NetChannelEnd.createNet2One();
85      return new Process2One(netIn);
86    }
87    protected static Process2One createNet2One(String channelName){
88      NetAltingChannelInput netIn = CNS.createNet2One(channelName);
89      return new Process2One(netIn);
90    }
91    protected static Process2One createNet2One
92                                        (NetAltingChannelInput netIn){
93      return new Process2One(netIn);
94    }
95    ...  plus other variants
96  }
```

The class `Process2One` references the `DynamicClassLoader` (`theClassLoader`) created in the singleton class `Mobile`. From `theClassLoader` it obtains the read filter {105}, which it attaches to the process channel {106}.

```
97  class Process2One extends AltingProcessChannelInput {
98    private FilteredAltingChannelInput filterIn;
99    private NetAltingChannelInput netIn;
100   private static DynamicClassLoader theClassLoader =
101                                          Mobile.theClassLoader;
102   protected Process2One(NetAltingChannelInput netIn) {
103     super(netIn);
104     this.netIn = netIn;
105     filterIn = FilteredChannelEnd.createFiltered(netIn);
106     filterIn.addReadFilter(theClassLoader.getChannelRxFilter());
107   }
108   ... plus some other methods not required for the discussion
109 }
```

The class `One2Process` undertakes the complimentary operation to that just described except that it adds a write filter to an output channel. An output channel can be created either using a `NetChannelLocation` {114}, a string reference to a Net Channel {119} or by using an existing Net Channel {124} as shown below.

```
110 class One2Process implements ProcessChannelOutput {
111   private NetChannelOutput netOut;
112   private FilteredChannelOutput filterOut;
113   private DynamicClassLoader theClassLoader = Mobile.theClassLoader;
114   protected One2Process(NetChannelLocation channelLocation){
115     netOut = NetChannelEnd.createOne2Net(channelLocation);
116     filterOut = FilteredChannelEnd.createFiltered(netOut);
117     filterOut.addWriteFilter(theClassLoader.getChannelTxFilter());
118   }
119   protected One2Process(String channelName){
120     netOut = CNS.createOne2Net(channelName);
121     filterOut = FilteredChannelEnd.createFiltered(netOut);
122     filterOut.addWriteFilter(theClassLoader.getChannelTxFilter());
123   }
124   protected One2Process(NetChannelOutput netOut){
125     this.netOut = netOut;
126     filterOut = FilteredChannelEnd.createFiltered(netOut);
127     filterOut.addWriteFilter(theClassLoader.getChannelTxFilter());
128   }
129   ... plus some other methods not required for the discussion
130 }
```

1.4 MobileProcess

`MobileProcess` is an abstract class {131-156} that is extended by a concrete implementation of a class that is to be made mobile. The class has a number of methods that permit the connection of arrays of input and output channels that the mobile process can use. There are a number of such methods that permit a variety of operation on the channels, enabling the attachment and detachment of channels dynamically.

```
131 public abstract class MobileProcess implements
132                                          CSProcess, Serializable {
133    protected ChannelInput[] in = null;
134    protected ChannelOutput[] out = null;
135
136    public final void init(ChannelInput[] in, ChannelOutput[] out) {
137      this.in = in;
138      this.out = out;
139    }
140    public final void remove() {
141      this.in = null;
142      this.out = null;
143    }
144    public final void attachInputChannels(ChannelInput[] in) {
145      this.in = in;
146    }
147    public final void attachOutputChannels(ChannelOutput[] out) {
148      this.out = out;
149    }
150    public final void detachInputChannels() {
151      in = null;
152    }
153    public final void detachOutputChannels() {
154      out = null;
155    }
156 }
```

1.5 Mobile Process Server

`MobileProcessServer` is the process used by an application's server to send mobile processes to the client processors. The application's server will in fact be the parallel composition of the `MobileProcessServer` and a sender process. The sender process creates the Mobile Processes and when required outputs these to the MobileProcessServer, which then sends them over the network to a client process. The MobileProcessServer has no knowledge of the process it is sending.

`MobileProcessServer` inputs the mobile process from sender on the internal channel `processIn` {180}. The mobile process is output on the `ProcessChannelOutput processOut` {181}, which will have all the required filters and access to `theClassLoader` automatically provided. The `NetChannelInput processRequest` {160} is used by the `MobileProcessServer` to receive requests from client processes for a mobile process. The `MobileProcessServer` is implemented as a singleton with an empty constructor.

The `init` method {169-172} is passed the name of the network channel, `serviceName`, upon which it will receive requests for client processes and uses this name to create the only named network channel in the system. The `init` method is also passed the internal channel by which it is connected to the sender process. In due course, the process's run method is invoked by the application's server (see 2.1.1 for an example description).

The run method reads the location of a network input channel {177} that will read the mobile process from the server using the previously created `processRequest` net channel. The `MobileProcessServer` then creates {179} an anonymous network output channel using a call to the `Mobile` class, which connects the `MobileProcessServer` to the `MobileProcessClient` that requested the mobile process. The server then reads {180} the `MobileProcess` from the sender using the internal channel, `processIn`, and then writes `theProcess` to the network channel connected to the client processor {181}.

```
157 public class MobileProcessServer implements CSProcess {
158   private ChannelInput processIn;
159   private ProcessChannelOutput processOut;
160   private NetChannelInput processRequest;
161   private static MobileProcessServer theServer =
162                                       new MobileProcessServer();
163   private MobileProcessServer()
164   {
165   }
166   public static MobileProcessServer getServer() {
167     return theServer;
168   }
169   public void init(String serviceName, ChannelInput processIn) {
170     processRequest = CNS.createNet2One(serviceName);
171     this.processIn = processIn;
172   }
173   ... some other access processes
174
175   public void run() {
176     while (true) {
177       NetChannelLocation clientLocation =
178                           (NetChannelLocation)processRequest.read();
179       processOut = Mobile.createOne2Net(clientLocation);
180       MobileProcess theProcess = (MobileProcess)processIn.read();
181       processOut.write(theProcess);
182     }
183   }
184 }
```

1.6 Mobile Process Client

The `MobileProcessClient` is the only process that has to run on a client processor and is typically invoked by a call to `main()`. An extension does allow the process to be invoked as part of a parallel compilation if the client side needs to receive more than one process or is itself part of another network. The process invokes itself and then enters the `main()` method. The user {195} is asked for IP address of the CNS applicable to the network and the name {196} of the network channel by which the client process will communicate with the server process. This must be exactly the same character string as used by the server process in its `init`() method. These values could be passed as arguments to the `main()` method if the code were modified appropriately.

The client process initializes itself as part of a mobile system {200} and connects itself to the same network by means of identifying the IP address of the CNS Server. The client then uses `CNS.resolve` {202} to connect itself to the CNS. This is used so that many clients can use the same network channel in sequence. Once the network channel has been resolved the connection to the server can be created as a net output channel {203}. The client then creates a net input channel (`processReceive`) by means of a call {205} to the static method `Mobile.createOne2Net`. The location of this channel is then communicated to the server by writing its channel location to the server {206} using the only named

network channel that was previously resolved. The mobile process is then read {207} from the anonymous network channel `processReceive`. The received mobile process is then executed by means of constructing a `ProcessManager` and thereby invoking the mobile process `run()` method {208}.

```
185 public class MobileProcessClient implements CSProcess {
186   private static SharedChannelInput serviceNameInput;
187   private static ProcessChannelInput processReceive;
188   private static SharedChannelOutput processOut;
189   private static MobileProcessClient theClient =
190                                          new MobileProcessClient();
191   private MobileProcessClient()
192   {
193   }
194   public static void main(String[] args) {
195     String CNS_IP = Ask.string("Enter IP address of CNS: ");
196     String processService =
197                       Ask.string("Enter name of process service: ");
198     try
199     {
200       Mobile.init(Node.getInstance().init(
201                                       new TCPIPNodeFactory(CNS_IP)));
202       NetChannelLocation serverLoc = CNS.resolve(processService);
203       NetChannelOutput toServer =
204                               NetChannelEnd.createOne2Net(serverLoc);
205       processReceive = Mobile.createNet2One();
206       toServer.write(processReceive.getChannelLocation());
207       MobileProcess theProcess =(MobileProcess)processReceive.read();
208       new ProcessManager(theProcess).run();
209     }
210     catch (NodeInitFailedException e)
211     {
212       System.out.println("Failed to connect to CNS");
213       System.exit(-1);
214     }
215   }
216   ...  plus some other access methods and a run()
217 }
```

2. Using `jcsp.mobile`

2.1 Maze Game

The maze game is itself quite simple and can be played by up to four players. Each player receives a copy of the same random maze. The goal of the game is to move the player's token from one corner of the maze to the opposite corner. The effects of each player's moves are shown on a central display. The players can only see their own token on their display. Because the maze is generated randomly it is possible that a player could be completely blocked in by walls. Thus each player is given the ability to clear a limited number of walls in order to make progress to their goal. As players move they can erect walls in their wake to block their opponents. The user interface provides buttons for moving in all four directions. Additionally, there are two further buttons that permit the clearing of a wall or the erection of a new blocking wall. As the players achieve their goal they are informed of their finishing position.

The sequence of interactions between the processes is as follows. The display process acts as the server. It creates the random maze and then sends this maze to each of the

players indicating their starting position and hence implicitly their goal. Thereafter, the player process sends the moves the player has requested to the display process. The display process obtains a move from each player at every step regardless of whether the player has made a move or not. The display process then updates its maze accordingly and then communicates the new state of the maze to each player process. If the display process detects that a player has reached their goal an appropriate indication is sent to that player only. This set of channels is captured in Figure 3.

In constructing the system as a set of mobile processes, the input ends of channels are created first, so that the input end location can be sent to processes that output on a network channel. Hence the `Display` process will create all the input ends of the `fromPlayer` channels, which will be communicated as part of the mobile `Player` process to each of the players. The `Player` process will then create the input ends for the `init` and `toPlayer` channels, which it will send back to the `Display` process using the `fromPlayer` channel. All these channels are created anonymously, which removes the complexity of creating a universal naming convention for mobile systems.

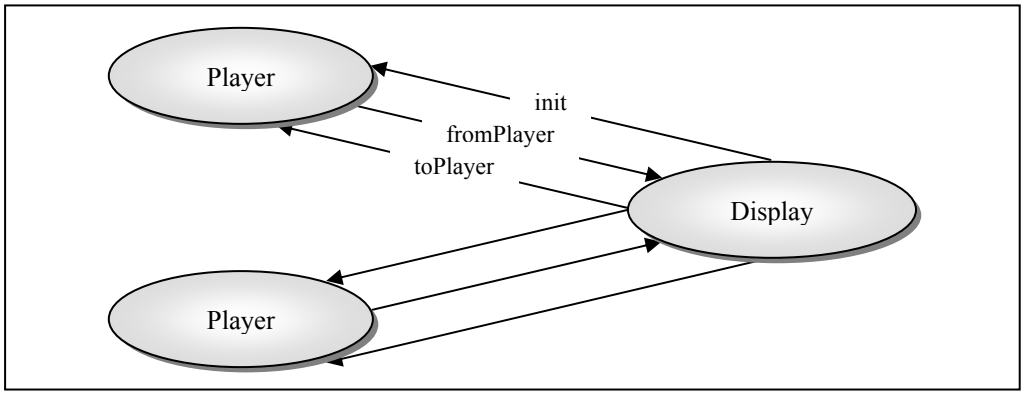

Figure 3 Network Channel Structure for the Maze Game

2.1.1 MazeServer

The `MazeServer` process is the main entry point on the server side of the system. It is used to implement the process `Display` in Figure 3. The user is asked for the CNS Server's IP address {226} and also the name of the service required {227}, which must be the same as that specified in the `MobileProcessClient` processes that request and then run each of the `Player` processes shown in Figure 3. Additionally, parameters that specify the number of players and the size of the maze are requested {228-230}.

The server process then initializes itself as `Mobile` connected to the identified CNS Server {231}. The channel (`toServer`) that connects the server to the sender is then declared {233}. The `sender` process is then declared {234} as an instance of `MazeServer` and its constructor {247-253} is passed the output end of the `toServer` channel and the parameters of the game. The server process is then declared {236} as an instance of `MobileProcessServer` and initialized {237} with the input end of the `toServer` channel and the name of the service that is provided. Recall that this `serviceName` is that of the only network channel that is created using a character string to identify the channel and provides the initial connection between any client and the server. A Parallel is then constructed {238} comprising `theServer` and `sender` and their run methods invoked.

The run method of the `sender` process is then defined as part of the `MazeServer` class {255}. An array of `ProcessInputChannel`s is declared that implements the `fromPlayer` channels {256}. An array of `MobileMazePlayer`s (see the next section) is then declared {258}. For each of the players, a `MobileMazePlayer` is defined with the required parameters, the identifier of the player and the channel location of the `fromPlayer` net input

channel {259-263}. As requests are received from the client processes the next mobile player process is sent to the client process. Thus a client does not know which player process they will receive.

Once all the player processes have been sent to each of the client processes an instance of the `MazeDisplayMain` process is declared {265} and executed by passing it as a parameter to the `ProcessManager` constructor {267}. We shall not describe the detail of the `MazeDisplayMain` process but its first stage is to read in the net channel locations of the input ends of the `init` and `toPlayer` channels.

```
218 public class MazeServer implements CSProcess {
219   private int noPlayers;
220   private ChannelOutput out;
221   private int rows, columns;
222   public static void main(String[] args)
223   {
224     try
225     {
226       String CNS_IP = Ask.string("Enter IP of CNS: ");
227       String serviceName = Ask.string("Enter unique service name: ");
228       int noPlayers = Ask.Int("Enter number of players: ", 1, 4);
229       int rows = Ask.Int("Rows: ", 5, 20);
230       int columns = Ask.Int("Columns: ", 6, 20);
231       Mobile.init(Node.getInstance().init(
232                                   new TCPIPNodeFactory(CNS_IP)));
233       One2AnyChannel toServer = Channel.createOne2Any();
234       MazeServer sender = new
235                MazeServer(toServer.out(), noPlayers, rows, columns);
236       MobileProcessServer theServer =MobileProcessServer.getServer();
237       theServer.init(serviceName, toServer.in());
238       new Parallel(new CSProcess[] {theServer, sender}).run();
239     }
240     catch (NodeInitFailedException e)
241     {
242       System.out.println("Error connecting to CNS");
243       System.exit(-1);
244     }
245   }
246
247   private MazeServer(ChannelOutput out, int noPlayers,
248                      int rows, int columns) {
249     this.out = out;
250     this.noPlayers = noPlayers;
251     this.rows = rows;
252     this.columns = columns;
253   }
254
255   public void run() {
256     ProcessChannelInput[] fromPlayers = new
257                                   ProcessChannelInput[noPlayers];
258     MobileMazePlayer[] players = new MobileMazePlayer[noPlayers];
259     for (int i = 0; i < noPlayers; i++) {
260       fromPlayers[i] = Mobile.createNet2One();
261       players[i] = new MobileMazePlayer(i,
262                             fromPlayers[i].getChannelLocation());
263       out.write(players[i]);
264     }
265     MazeDisplayMain theMaze = new MazeDisplayMain(noPlayers,
266                                     fromPlayers, rows, columns);
267     new ProcessManager(theMaze).run();
268   }
269 }
```

2.1.2 MobileMazePlayer

The `MobileMazePlayer` class extends `MobileProcess` and contains a constructor {273} that is invoked by the `MobileProcessServer` when the instances of the mobile process are created. The `MobileProcessClient` that inputs this process invokes the run method {278-287}. The run method declares the `init` and `toPlayer` channels {279, 282} using a call to `Mobile.createNet2One`. The location of the `fromPlayer` channel is passed as parameter of the constructor {280}, which is used to create a `Mobile One2Net` channel. The mobile player process then writes the locations of the `init` and `toPlayer` channels to the Display process as described above using the `fromPlayer` channel {283,284}. An instance of a `PlayerFrame` process is then run {285}, which has not been modified in any way from the version that ran in a non-mobile manner.

```
270 public class MobileMazePlayer extends MobileProcess {
271   private int playerId;
272   private NetChannelLocation fromPlayerLocation;
273   public MobileMazePlayer(int playerNumber,
274                           NetChannelLocation fromPlayerLocation) {
275       this.playerId = playerNumber;
276       this.fromPlayerLocation = fromPlayerLocation;
277   }
278   public void run() {
279     NetChannelInput init = Mobile.createNet2One();
280     NetChannelOutput fromPlayer =
281                             Mobile.createOne2Net(fromPlayerLocation);
282     NetChannelInput toPlayer = Mobile.createNet2One();
283     fromPlayer.write(init.getChannelLocation());
284     fromPlayer.write(toPlayer.getChannelLocation());
285     new PlayerFrame(playerId, 100, init, fromPlayer, toPlayer).run();
286   }
287 }
```

2.2 Home Systems Remote Control

Within a current home, with many different electrical consumer products (TV, DVD Player, Stereo), there may be many different remote controls in use. The user may find it hard to remember exactly which control operates which product, they may lose a control or they may have too many controls in general. Using the mobile process technology, it is possible for an electrical product to contain its own remote control internally, and send it to a universal, touch screen device when requested. In other words, there is one control for all devices. This would effectively remove the requirement of product manufacturers developing remote controls for each device they create in a physical form, only a software developed system is required. This could be implemented using Bluetooth TCP/IP, or JCSP could be expanded to use serial connections and thereby allow the use of an infra red port.

Looking at digital television technology set top boxes currently download new system software when sent via the communication medium. Applying the concept of internally stored remote control systems means that they can also update their remote controls. This allows systems to be updated in ways not normally possible and coupling this with the ease of buying a new remote control, when one is lost, demonstrates the possibilities of these systems.

To demonstrate, a simple remote control device system was developed. Three devices that can be controlled are created, and are virtually demonstrated as three different windows on a PC. The remote control for each device will be transferred, as a Mobile Process, to a

PDA when requested. To simplify the concept, the PDA has an initial interface consisting of three buttons named TV, DVD and STEREO. When one of these buttons is pressed, the relevant control is received from the relevant device server. Figure 4 illustrates the basic process network design and shows how it relates back to the pattern presented for mobile processes.

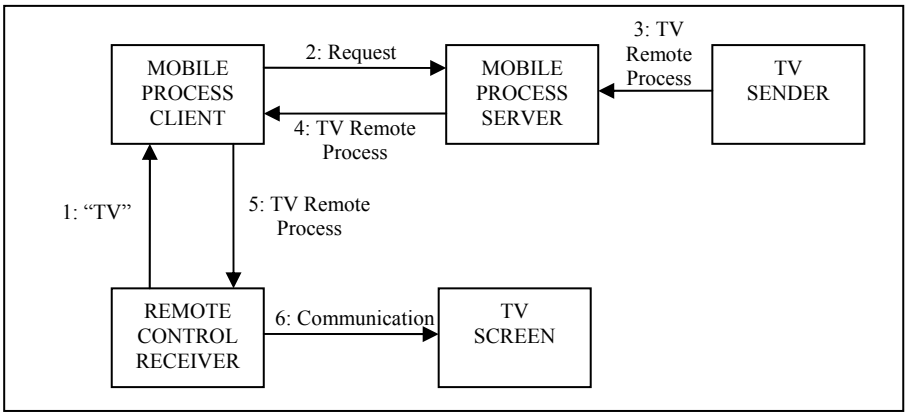

Figure 4 Remote Control System Process Network

3. Migrating Processes and Channels

The first step here is to modify the MobileProcess so it reflects more closely the model defined by Barnes (Barnes, 2001) by adding a channel that can be used to signal the process to stop. Originally this was called a kill channel, but reset is probably a more easily interpreted name, as the process is not necessarily killed. This channel can be used to send a signal to stop the process, or receive a channel location so that the process can be redirected to a new Node.

3.1 Migratable Channels

A Migratable Channel is one that can move from one Node to another and continue communication with its opposite channel end almost seamlessly. At a local level, this type of operation is easy with Java as a reference to a Channel End can be quite easily passed from process to process. Doing this at a distributed level is a lot more difficult, as it requires the opposite channel end to be redirected to the new location.

A Networked Channel operates using client-server style architecture, with the input end acting as a server, and the output as a client. Because of this it is possible to have multiple output ends (clients) connected to a single input end (server), and all the output end needs to know is the location of its relative input end. If this is viewed in the context of a Migratable Channel, then as long as the migrating output channel retains the location of the input channel it is using, it can be quite easily sent across a network.

Input channels are a bit more difficult. If it is moved to another location then each of its relative output ends must be made aware of this. Due to the input channel having no knowledge of its output channels, the obvious solution of sending the new location to these channels is not possible. Quickstone have developed a solution to this problem however.

What appears to happen is that when a Migratable Input Channel is moved, it leaves a small process listening on its original location with a temporary channel name it declares with the CNS. When the input channel arrives at its new location, it declares this channel with the CNS. When an output channel then tries to send to the old location, it is told the name of the channel to resolve with the CNS, which it then uses to reconnect to the channel. Figure 5 illustrates this concept.

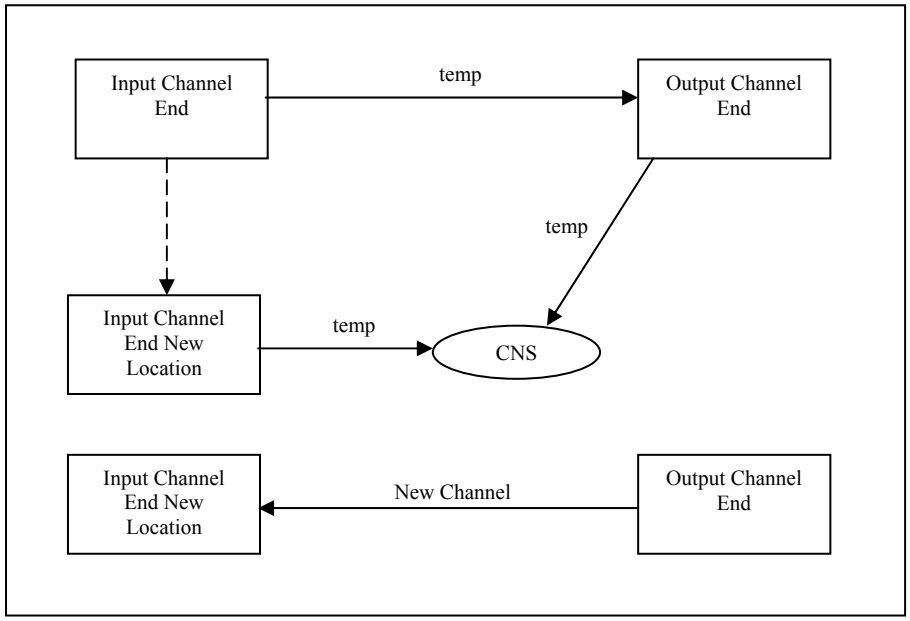

Figure 5 Migrating Channels

This method works quite well, as long as both Channel Ends are declared as Migratable. This is so that the output end knows in some cases it may receive a name to connect to a new channel location. In both instances (input and output), when the channel is about to move, a prepareToMove() method is called on the channel so that any necessary actions can be performed on the channel as required before movement. This idea can be taken further into the concept of a Migratable process also.

3.2 Migratable Processes

If the Mobile Process class is extended to create a Migratable Process class, it can be seen that implementing a `prepareToMove()` on the process is also a good idea. This will allow any necessary actions to be performed on the process to prepare it to be moved (a prime example of this is the removal of non-serializable objects as these cannot be sent across networked channels). This method is declared abstract as each specific process will require different actions to prepare it for migration.

It is possible to test a basic Migratable Process, and the simplest to use is a Producer-Consumer system. Due to the simplicity of the system itself, it is not necessary to use the `prepareToMove()` function as no real preparation is necessary to move the process. It is also easier to have the basic processes wrapped inside a parent Migratable Process which will start them when it arrives at its relevant client location. This parent class will then read from its reset channel and when it receives a channel location will stop the Producer or Consumer, create a channel from the channel location received and write itself down the created channel.

3.3 A Producer-Consumer Example

When this version of the Consumer is created, it is passed a `MigratableChannelInput` end, which can be passed in when the server process creates it. This is similar to the idea of passing in a channel location but a channel does not need to be created by the process as it is already present. Due to the nature of the Migratable Channel this is perfectly viable code.

When the process is run, it initially declares the `reset` channel with the CNS {294}, which allows other process to find the reset channel. A new `consumer` process is then created {296}, and started within its own `ProcessManager` {297}. A `ProcessManager` allows a single process to be started and for the main process to continue if the `start()` method is called instead of `run()` {298}. Next a channel location is read from the declared `reset` channel {299}. When a location is received, the consumer is stopped and the `MigratableChannelInput` is informed that it is about to move {300-301}. A `ProcessChannelOutput` is then created {302} and the `MigratableConsumer` is sent down it {303}. When another client receives the process, this run method is invoked again with the Producer having no knowledge that the process has relocated, and hence the Consumer is truly mobile. The Consumer's input channel can also be made a Process Channel {295}, allowing unknown objects to be sent from the Producer to the Consumer.

```
288 public class MigratableConsumer extends MigratableProcess {
289   MigratableChannelInput in;
290   public MigratableConsumer(MigratableChannelInput in) {
291     this.in = in;
292   }
293   public void run(){
294     reset = Mobile.createNet2One("consumer.reset");
295     ProcessChannelInput processIn = Mobile.createNet2One(in);
296     Consumer consumer = new Consumer(in);
297     ProcessManager pm = new ProcessManager(consumer);
298     pm.start();
299     NetChannelLocation toLocation = (NetChannelLocation)reset.read();
300     pm.stop();
301     in.prepareToMove();
302     ProcessChannelOutput toNext = Mobile.createOne2Net(toLocation);
303     toNext.write(this);
304   }
```

4. Evaluation

4.1 Embedded Systems

Some of the main issues relating to embedded systems are memory constraints and the ability to update the systems easily. In general, an embedded system would require all the necessary class files and resource files before it could be started, and this can be a strain on storage requirements, especially if some of the class files are rarely used. If the framework for mobile processes is used, as well as a system that removes unnecessary class definitions from the Class Manager, the embedded system only requires the basic classes used here, and this will possibly lead to a smaller memory footprint. Of course, if time constraints are placed on the embedded system then this method is inefficient.

The other problem associated with embedded systems - that of updating multiple systems easily - can be implemented quite easily with this framework, and this can even be done without stopping the systems. By using the reset channel, it is possible to make the system update itself as necessary by forcing them to access the server process responsible for the clients.

4.2 Distributed Systems and Mobile Agents

The concept of a mobile agent was first discussed by White as a successor to Remote Procedure Calling [6]. White points out that the main limitation of RPC for distributed

systems is that the client requires some knowledge as to the service provided by the server, and describes the concept of mobile agents to overcome this limitation by allowing a system to inform a client type application exactly what kind of procedures it provides.

Looking at the framework developed here for mobile processes, it is apparent that a system that exhibits this kind of functionality has been created. The client side process does not need any prior knowledge as to what functionality it will have. In effect it is told exactly what procedures it can use when it requests a service.

Another concept of mobile agents is the ability to send a process from one system to another so it can execute some functionality and then return. Again, it is quite apparent that the mobile process framework also provides this functionality quite concisely. It is quite possible to send processes between systems easily, without either system knowing exactly what the process needs to do. Of course, there are security constraints that need to be considered to stop malicious processes being sent into a system, but the basic principle has been developed.

4.3 Location Aware Computing

Currently, location awareness within systems relates mainly to the availability of different information being available to the user depending on their current location, coupled with the ability for users to know each other's location. Basically the system adapts to offer different information and services relating to where the user happens to be.

An example of this type of environment can be seen in a museum, where currently exhibit information is generally contained on a display card next to the exhibit in question. This may require the onlooker to move away from the exhibit to the card to read information pertaining to the exhibit, and then back. If a location aware system was implemented in this situation, these cards could be replaced by information displayed on a PDA relevant to the exhibit the user is near. Furthermore, it would be possible for the system to provide links to other relevant information about the exhibit within the PDA, or externally on a server, with these links possibly being locations of other exhibits, audio, video or textual information.

This kind of system is being placed as the next generation of application for mobile devices, but the limitation of providing information relevant to the location is apparent, and is really only an extension on current browser technology. What the mobile process technology has shown is that it is possible to change the system itself, to a Location Adaptive System or Location Context System, where the system itself adapts to the location of the user.

5. Conclusion and Future Work

The first point to consider when evaluating JCSP is how well it reflects the underlying principles of Communicating Sequential Processes, and although no thorough investigation into this has taken place here, it can be deemed by some of the resources used that JCSP does provide the mechanisms expected [7,8]. Using JCSP is relatively easy, and can be related back to the basic premise of individual processes sharing resources by the use of communication channels. JCSP, or more importantly Java, does present some issues relating to the implementation of a CSP system at a local level, due to the reference semantics used. This means even though an object is passed from one process to another over a channel the sending process can still change the shared object [3].

A more accurate evaluation can be made of the networked aspects made available by JCSP Network Edition, especially in reference to the robustness of the Channel Name

Server and underlying architecture used to implement the networked aspect. During this work, the CNS has been tested on a number of levels. Firstly, the ease of creating networked channels given only the IP address of the CNS and a channel name should be mentioned. This feature creates an almost invisible method of networked channel creation, with locations of individual processes not being required, just the location of the CNS they are using. This functionality can be attributed to the implementation of the Broker design pattern [9].

The second feature of the Network Edition of JCSP is anonymous channels. This functionality is one of the main reasons that the concept of mobile processes within JCSP is possible. Using anonymous channels, by passing channel locations within a process, another layer of transparency is possible, mainly as the CNS does not have to be used to forward messages. This is where the CNS advances beyond the original Broker pattern used. In theory it is quite possible to have a system that does not use a CNS after an initial process transaction as it is never used again. This concept is also the reason why the Dynamic Class Loading service can be made available, as a process can be given the location to retrieve class data from, again transparently. The whole transparency and ease of use of these features truly demonstrates how well the networked features have been developed.

The final feature of consideration is how the channels have been developed. There were two choices on how to develop a networked channel, having the underlying server socket as an input channel end receiving messages, or have the output channel acting as the server socket. Given that servers generally send information to clients it would seem most likely that the output channel would act as a server socket. However, had this type of architecture been implemented, it would not be possible to send channel locations between distributed processes as the initially declared output channel would only be able to send locations, not receive them. The designers of JCSP Network Edition took this into consideration however, and developed the input channels as server sockets, enabling the development of mobile processes as a whole.

Also due to the good development was the ease with which JCSP could be transferred onto a mobile platform, although this was mainly because of the underlying Java Runtime Environment used. A viable version of J2ME Connected Device Configuration Personal Profile, which acts as Java Standard Edition 1.3.1, was made available by IBM[10]. It was fairly easy to transfer a basic JCSP system onto a PDA device and it is even possible to use a PDA device as the CNS.

Currently, we are developing a system for context and location aware computing that will allow a user to roam around an environment such that their PDA will download location specific clients to interact with a server depending upon the particular wireless access point they are using to access the network.

Acknowledgements

Kevin Chalmers acknowledges the support of the Student Award Agency Scotland who contributed to his tuition fees for the undergraduate degree for which the work reported in this paper formed part of his final year project. We are grateful for the helpful comments of the referees.

References

[1] Quickstone Ltd, web site accessed 4/5/2005, http://www.quickstone.com/
[2] Barnes, F.R.M & Welch, P.H. (2003, April). *Prioritised Dynamic Communicating and Mobile Processes*. IEEE Proceedings – Software, 150(2), 121-136.
[3] Barnes, F.R.M. & Welch, P.H. (2001). *Mobile Data, Dynamic Allocation and Zero Aliasing: an occam Experiment*. In Chalmers, A., Mirmehdi, M. & Muller, H. (Eds.), Communicating Process Architectures 2001. IOS Press.
[4] Qian, Z., Goldberg, A. & Coglio, A. (2000, October). *A Formal Specification of Java Class Loading.* ACM SIGPLAN Notices, 35(10), 325-336.
[5] Welch, P. (2002). *CSP Networking for Java (JCSP.Net).* [Electronic Version]. PowerPoint presentation. Retrieved January 14, 2005 from http://www.cs.kent.ac.uk/projects/ ofa/jcsp/jcsp.ppt.
[6] White, J. (1996). *Mobile Agents White Paper*. [Electronic Version]. General Magic. Retrieved March 22, 2005 from http://citeseer.ist.psu.edu/white96mobile.html.
[7] Welch, P.H. (1998). *Java Threads in the Light of occam/CSP.* In Welch, P.H. & Bakkers, A.W.P. (Eds.), Architectures, Languages and Patterns for Parallel and Distributed Applications, WoTUG-21 (pp 259-284). IOS Press.
[8] Welch, P. & Martin, J.M.R. (2000). *Formal Analysis of Concurrent Java Systems.* In Welch, P.H. & Bakkers, A.W.P. (Eds.), Communicating Process Architectures 2000 (pp 275-301). IOS Press.
[9] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. & Stal, M. (1996). *A System of Patterns: Pattern-Orientated Software Architecture.* Chichester: Wiley.
[10] IBM J2ME, web site accessed 4/5/2005. http://www-106.ibm.com/developerworks/wireless/library/wi-j2me/